Policy-driven tailoring of sensor networks

Nelson Matthys[1], Christophe Huygens[1] Danny Hughes[2],
Jó Ueyama[3], Sam Michiels[1], and Wouter Joosen[1]

[1] IBBT-DistriNet, Katholieke Universiteit Leuven, 3001 Heverlee, Belgium,
{firstname.lastname}@cs.kuleuven.be
[2] Department of Computer Science and Software Engineering,
Xi'an Jiaotong-Liverpool University, 215123 Suzhou, China
daniel.hughes@xjtlu.edu.cn
[3] University of São Paulo (USP), São Carlos, 13566-585, Brazil
joueyama@icmc.usp.br

Abstract. The emerging reality of wireless sensor networks deployed as long-lived infrastructure mandates an approach to tailor developed artefacts at run-time to avoid costly reprogramming. Support for dynamic concerns, such as adaptation, calibration or tuning of the functional and non-functional behaviour by application users and infrastructure managers raises the need for fine-grained run-time customization. This paper presents a policy-based paradigm to realize the diverse concerns of the involved actors by enabling fine-tuning and optimization of the runtime environment. Integration of the policy paradigm into various main programming models is analyzed. A prototype implementation of the paradigm in the context of an event-component based wireless sensor network platform is evaluated on the SunSPOT sensor platform.

Key words: Policy, Component models, Reconfiguration, Multi-paradigm Programming

1 Introduction

Over the last few years, Wireless Sensor Networks (WSN) have evolved into long-lived infrastructure on which various applications from multiple actors may be executing concurrently [25, 24, 21]. This trend of moving away from the traditional monolithic application paradigm towards a general purpose execution platform capable of hosting a multitude of applications has already been exemplified by several scenarios that explore the advantages of using WSNs, such as environmental monitoring [16], road monitoring [4], or logistics [21]. In these application scenarios, WSN devices play a role which is merely not data-centric but expands to the execution of a localized part of the holistic application, in which the sensor acts as a general purpose execution platform, albeit with limited execution capabilities. As such, WSN infrastructure is becoming another tier of enterprise infrastructure on which various software components can be deployed over time, and which are potentially used and administrated by different actors.

Both these multi-actor and long-lived usage modes drive the need for run-time customization or adaptation. More specifically, the need for run-time adaptation can be due to (i) changing requirements of a single application end-user in a long-lived setting, since it is impossible to capture all (future) requirements at development time, (ii) driven by the benefits of reuse by running customized versions of a single application for various application users, and (iii) plain change in the system environment over time. In the first case (i), the demands of the user with respect to the system change: a sampling frequency sufficient today may no longer be appropriate tomorrow. This ultimately comes down to the same requirement raised by the multi-actor perspective (ii) where services may differ only through customization of service behaviour, for instance in sampling frequency, persistence or security of collected data. These objectives are typically very specific to each user, and therefore cannot always be fixed during platform development time. To illustrate (iii), it is straightforward to envision different behaviour depending on location or energy status. As a result, appropriate programming abstractions must be foreseen that allow for each individual actor to tailor its portion of business functionality over time at run-time.

In this paper, we propose a policy-based abstraction and associated programming model to accommodate the adaptation requirement, hereby enabling variations in functional and non-functional concerns of application end-users and system administrators. We demonstrate how our policy-driven system integrates in a non-intrusive way with (the interaction models) of the application logic programmed in the WSN and how it operationally aligns with the activities of the WSN administrative stakeholder. A prototype implementation on the SunSPOT [23] sensor platform is presented enabling flexible and fine-grained customizability of the WSN, while respecting its resource-constrained nature in terms of memory footprint and performance overhead. We validate through a case study in a WSN-based flood monitoring application where the policy-driven system is used to tailor the individual concerns of the various actors.

The remainder of this paper is structured as follows. Section 2 motivates the case for and highlights the requirements of fine-grained customization in long-lived WSN scenarios involving multiple actors. Section 3 discusses how the policies are used as a paradigm to express and support the various concerns of different stakeholders and how to integrate them in existing systems. Section 4 validates a prototype implementation through a flood monitoring case, while its performance and overhead is evaluated in Section 5. Section 6 discusses related work. Finally, Section 7 concludes and sketches future work.

2 Fine-grained customization in multi-actor WSNs: motivation and requirements

WSNs are moving away from the monolithic application model towards a shared-infrastructure, multi-application usage mode where every device has an administrative owner controlling the device [10]. In this setting, multiple applications

reflecting the functional (business) goals of their respective owners may execute concurrently on a node [25, 21].

Consider for instance an environmental river-monitoring scenario where multiple entities may leverage a WSN to gather temperature, pollution, and water level data. Both governmental agencies as well as ecological scientists from universities are interested in pollution and temperature data, whereas local government is interested in flooding data to protect the livestock of the community along the river. During commissioning, certain activities are executed: first and foremost the application is composed out of subprograms. These programs are then deployed. Finally an initial configuration defining fixed sensor sampling and reporting rates is established. From this scenario, we can identify three drivers for run-time customization:

1. Since at commissioning time, ecologists do not know what base pollution values to expect, the alerting threshold will need updating. In fact, this value will change frequently with evolution of water quality. *Within a long-lived application, customization happens frequently reflecting tuning or changing requirements.*
2. The data gathering activities of the government and the scientist are fundamentally similar yet subtle differences in level of detail need to be accommodated. Large efficiency gains (footprint, programming effort) can be made in the WSN by using customized variants of the same application, for example with different operational parameters. *Within a multi-actor setting, the possibility to customize greatly facilitates reuse and resource efficiency.*
3. Alerting thresholds need to be adjusted depending on the location of the sensor and time of season, for example in extreme rainfall events pollution will typically be very high because of sewerage overrun. *Customization rules describe change in system behaviour in response to system dynamics.*

Due to dynamism in many of these scenarios, WSN customization is not solely a one-time process as previous real world WSN deployments [16] have revealed, but rather a continuous process. Many operational parameters of the WSN (or its constructing system or application parts) will change continuously in response to changes in objectives of applications users, system administrators or in response to internal change such as mobility or available energy. In addition, the change cycle of above customizations is closely related to the type of abstraction which is used to realize that particular part of the application that is impacted by this change. For instance, pure business functionality, such as a generic sensing component, is changed less often compared to the rules governing the sampling behaviour of that component during extreme rainfall.

Finally, since the above customizations are performed by both application end-users or system administrators during the run-time phase of the application, it should be recognized these users prefer to express their goals differently than developers [2]. Abstractions offered for run-time customization must therefore be intuitive and sufficiently high-level. In this context, rule-based declarative or imperative approaches are typically considered to be appropriate abstractions [10, 3].

From this, the requirements of the mechanism to support these run-time customizations can now be identified:

1. Fine-grained: supporting small and specific changes in behaviour and as such complementary to the development and composition activities of the applications where behaviour is described at large.
2. Light-weight: since the changes are frequent, it is beneficial to the WSN that their representation is compact and their execution imposes little overhead.
3. End-user abstraction: adequate abstraction so that less technical end-users rather than expert WSN programmers can express their goals.

These requirements are in contrast with typical WSN reprogramming approaches such as TinyOS [15] that work monolithically, or even component-based engineering [8] where only coarse-grained change is supported. Component models do provide an attractive programming model for building multi-actor WSNs since new applications can be constructed in a cost-effective and resource-efficient manner by reusing existing components. Some level of support is offered in component models for the problems of dynamism and long-livedness of the WSN through the ability to add or delete functionality or modification of the existing composition at run-time. Yet, components are typically implemented as coarse-grained generic artefacts applicable for a wide variety of applications with little support for domain-specific customization. Thus, while component-based reconfiguration provides a generic mechanism for enacting changes, it is inefficient when only a few lines of code may represent that change. This is particularly critical for WSNs, where memory is limited and software updates are costly. So, while the component approach has clear merits, the combined requirements regarding change granularity, frequency, and end-user abstraction demand a specific solution with low development overhead, memory footprint, and performance overhead. Furthermore, the solution should integrate in a non-intrusive manner with the existing main development paradigm. Therefore, we introduce a lightweight framework for adapting application behaviour based on policies. Policies for this framework are high-level, declarative and platform independent, allowing end-users to easily tailor behaviour.

3 Policy as paradigm for fine-grained customization

Like regular programs, policies are abstractions that govern the behaviour of applications. By using policies functionally, the application actor can fine-tune the behaviour of a business function so that it better serves its purpose. Non-functionally, an infrastructure manager can realize its security or energy concerns through policy specification. For instance, sharing policies may indicate what actor may use which piece of functionality. In this section, we first discuss the life cycle of using policies as a programming paradigm for fine-tuning WSN behaviour. Secondly, we focus on possible strategies for integration in existing systems.

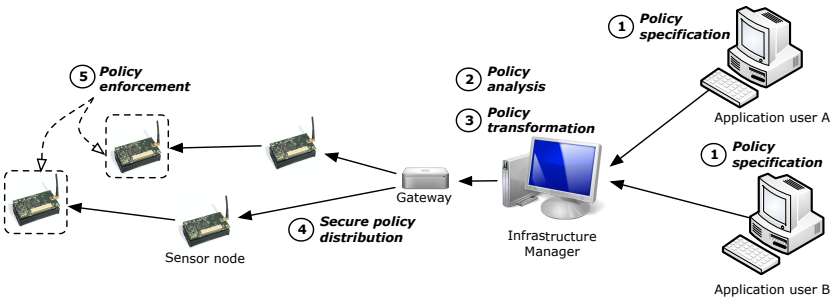


Fig. 1. Policy life cycle: specification, analysis, distribution, and enforcement

3.1 Policy life cycle

Several activities take place between policy specification by the application user and their actual enforcement or execution on the target device. Policies start their life cycle as high-level user-specified policies and subsequently undergo several levels of transformation and refinement to end up as code or configuration specifications that can be directly deployed and executed on the elements of an IT infrastructure [1]. To be applicable in the context of resource-constrained systems such as WSNs, it is crucial to make optimal use of the capabilities offered by the resource-rich back-end to perform most of these activities which are inherently heavyweight. Figure 1 illustrates the chain of activities that constitute the policy life cycle in the context of WSNs:

1. Policy specification happens by (non-technical) application users using tools helping them to create syntactically and semantically correct policies written in a policy language. These policies are then submitted to the administrative actor together with a list of applications and target nodes where they should be applied.
2. Policy analysis is a heavyweight activity which should exclusively happen in the resource-rich back-end of the administrative actor. It involves checking for conflicts between already applied policies and verification whether the policy is transparent for other users. If a policy is found to be correct, it is marked for transformation and distribution to the nodes.
3. Next, the policy is transformed into a more compact and optimized representation better suited for energy-efficient dissemination inside the WSN.
4. Since policies may potentially govern control over all types of functionality deployed inside the network, secure policy distribution is an essential part of the life cycle. Therefore, it is particularly critical that only authorized actors can deploy these policies and that they are disseminated in a secure and reliable fashion.
5. Installation and policy enforcement on a node happens upon reception and verification of the policy. After reception of the compact policy represen-

tation, an implementation-specific data structure more suitable for efficient evaluation is constructed which is then stored locally.

3.2 Integration strategies

Integration of the proposed policy-based programming paradigm with the main application development paradigm of the WSN should be transparent for the application developer whilst guaranteeing execution of the policies. At some point execution must be transferred from application code to policy code. The exact point in an application where redirection to the set of applicable policies or corresponding refined code is performed depends on the interaction model used for application composition. Classical interaction paradigms [5] in traditional distributed applications include amongst others (remote) procedure calls, method invocations, or event-based messaging, and the base entities (components in these systems) are considered reusable, solely identified by their type along with their interfaces and dependencies [8].

At development time, all applications are composed by combining individual components through a set of connections which model interface dependencies. At run-time, these connections are represented by a series of inter-component procedure calls or communication messages, depending on the interaction model used. Since both syntax and semantics of these component interactions are well standardized, it is advantageous to limit policy enforcement to these component-to-component interaction points. For tailoring and customization purposes of the application this set of points is sufficient. Hence, the integration of the policy-based paradigm must a priori be supported by the interaction model or must be inserted in the interaction model before deployment through selective instrumentation. Regarding the interaction model, since (remote) procedure calls and method invocation are resource-heavy, they are not suited for the WSN target environment as scalability is compromised [14]. Therefore, we limit the scope of our research to event-based messaging, which is commonly used as interaction model in WSNs [15, 18], and integrate our policy-based paradigm directly inside this interaction model.

3.3 Paradigm benefits

Policies offer an attractive development paradigm complementary to the main application development paradigm used inside the WSN system. First, they provide a powerful abstraction for additional customization as they allow to specify and enforce various functional and non-functional concerns at run-time. Secondly, since policies are focused on a single concern, they can be lightweight (see Section 4). Thus customization is achieved at a reasonable energy cost. Finally, the fine-grained, independent distribution model complements the update model used by the main development paradigm, since it supports small changes to application compositions, hence accommodating evolving application demands and dynamic environmental conditions.

4 Research prototype

A prototype of a framework supporting the proposed policy paradigm was implemented for an event-based run-time reconfigurable component model. For testing and evaluation purposes, we applied the resulting framework in the context of a small-scale real world river monitoring case.

4.1 Implementation

Base paradigm: component model. The Loosely-coupled Component Infrastructure (LooCI) [9] is a lightweight event-based run-time reconfigurable component model for WSNs. In the LooCI model, components are indirectly bound over an event bus abstraction, implementing a decentralized publish/subscribe interaction model. All LooCI components define their provided interfaces as the set of events that they publish, whereas the required interfaces of a LooCI component are similarly defined as the events to which it subscribes to.

Reconfiguration in LooCI is enacted by mechanisms to dynamically deploy, start, stop, or remove components together with dynamic re-wiring of component bindings. Defining a LooCI component implicitly provides access to the event bus for inter-component communications and the underlying connectionless network framework. As a result, all communication between LooCI components is carried by events that allow for asynchronous and indirect communication between a pair of components. For our research, the key benefits of LooCI are, along with a small footprint and good performance, that it promotes this event-based interaction paradigm and a loose coupling between cooperating components. The set of introspective facilities includes support to discover components and their bindings on a node or between two nodes.

Policy framework and language: The supporting framework for our policy based programming paradigm is illustrated in Figure 2. This framework is deployed on every node inside the WSN and is integrated with the LooCI event bus. Since all interactions between LooCI components occur as events over the event bus, it is possible to tailor multiple aspects of the system by modifying their content or configuring the manner in which these events are propagated. To facilitate this type of management, the LooCI run-time is extended with a compact policy engine, which executes a lightweight Event-Condition-Action (ECA) policy specification language. Every ECA policy consists of a description of the triggering events, a condition which is a logical expression typically referring to event criteria or external system aspects, and a list of actions to be enforced in response. Every time an event is sent between a pair of components via the event bus, the policy engine intercepts this event and evaluates how it should be modified based upon the set of per-node installed policy rules, stored inside a repository component. If the incoming event matches a policy rule, its associated action(s) will applied. At run-time, the set of policies can be dynamically updated accommodating evolving application requirements.

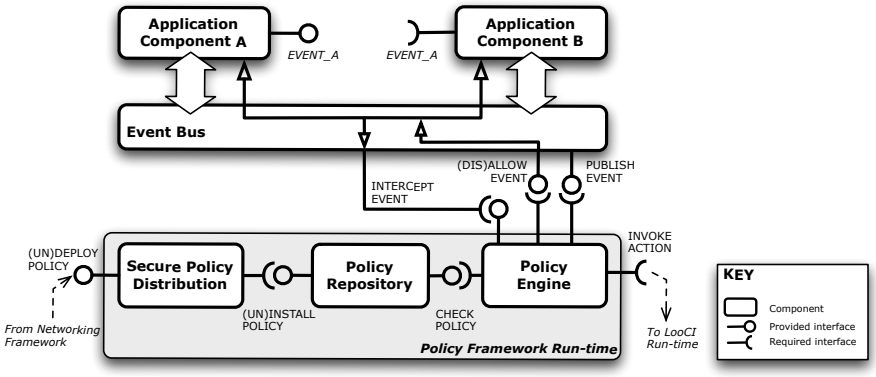


Fig. 2. Policy framework prototype

Our prototype policy language allows various functions to be invoked inside the condition and action parts of a policy. Amongst these include actions to allow/deny the event to pass, change its contents, replicate and reroute the event towards an intermediate component for additional processing, or to publish a custom user-defined event, for example containing a configuration value destined for a particular component. In this sense, policies offer a simple, yet powerful abstraction for end-users to fine-tune generic component behaviour, or for system administrators to inject various flavours of non-functional concerns.

Policy distribution model: As shown in Figure 1, the policy distribution model forces all policy administration to go through the gateway, which is under control of the infrastructure manager and has a direct trust relation with each sensor. In our research prototype, this trust between the gateway and the individual sensors is established by using a pre-deployed public/private key scheme.

Application users submit their (high-level) policies to the infrastructure manager in a secure manner using standard enterprise-grade security technologies such as TLS/SSL. Upon reception, these policies are analysed for consistency and transformed into a compact binary representation, more suitable for energy-efficient dissemination inside the WSN. In order to deploy a policy on a sensor node, the infrastructure manager then transfers this compact policy representation together with its associated deployment instructions to the gateway using a secure connection, ensuring authenticity and confidentiality of the said policies when distributed to the gateway.

Next, to securely deploy the policy inside the WSN, the gateway constructs a message M that securely encapsulates the following content D:

$$D = \texttt{[deploy\_instr, dest\_addr, id, timestamp, policy\_data]}$$
$$M = D, MAC(K_{GW}, D)$$

In this message, `deploy_instr` contains the deployment instructions such as install or remove a policy, `dest_addr` is the node where the policy needs to be

deployed, `id` indicates a sequence number used between the gateway and destination, `timestamp` is the current timestamp used as nonce, and `policy_data` is the binary representation of the policy sent to the WSN. For integrity purposes, a Message Authentication Code (MAC), denoted as $MAC(K_{GW}, D)$, of the policy data D is attached to the message M. This MAC is signed by the private key K_{GW} of the gateway.

Once received, the destination node can check whether the message has been sent from the gateway by using the pre-deployed public/private key pair to verify $MAC(K_{GW}, D)$. In this sense, non-repudiation of origin, integrity, and protection against replay attacks can be guaranteed. When this verification is successful, the `policy_data` can be extracted from the message, transformed by the policy engine into a data structure which is more suitable for efficient evaluation, and added to the list of policies installed on the node.

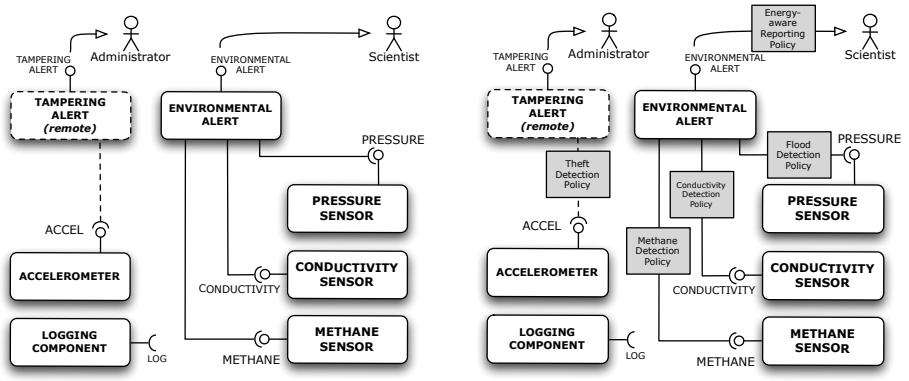
Note that this scheme only ensures integrity of the `policy_data` and does not provide confidentiality. Policy authenticity can be assured between each node and the gateway, and between the gateway and network administrator. Optionally, confidentiality can be ensured inside the WSN via symmetric encryption using a session key generated by the pre-deployed public/private key scheme.

4.2 Real world case study

The combination of a lightweight component framework LooCI and policy framework was evaluated in the context of a small-scale real world river monitoring case in the city of São Carlos, in São Paulo state, Brazil. In this scenario, the WSN consisted of four SunSPOT [23] sensor nodes that were deployed to monitor river water quality. Different local environmental science partners monitored three environmental factors over a two-week period: (i) water depth was monitored using a hydrostatic level sensor in order to provide early warning of flood events, (ii) water conductivity levels were monitored using a standard conductivity sensor in order to infer pollution levels, and (iii) methane levels were monitored using a simple CH_4 sensor in order to detect decaying organic matter. Finally, tamper and theft detection was implemented using the built-in three dimensional accelerometer of the SunSPOT.

Application composition: The river monitoring composition consisted of seven LooCI components that implemented generic functionality, including logging, encapsulation of hardware resources like sensors, or alert reporting. At platform commissioning time, these components were wired to each other as depicted in Figure 3(a).

- A pressure sensor component periodically polls the hydrostatic level sensor and exposes these readings through events of type 'PRESSURE'.
- A conductivity sensor component periodically measures water conductivity and exposes these readings through events of type 'CONDUCTIVITY'.
- A methane sensor component exposes readings from the CH_4 sensor via an interface of type 'METHANE'.



(a) Default component-based application composition

(b) Policy-augmented application composition

Fig. 3. River monitoring composition

– An environmental alert component processes these readings, adds additional context information such as node ID and timestamp, and forwards these readings to the environmental scientists.
– A generic logging component encapsulating access to the flash memory and processing 'LOG' events.
– Finally, to detect sensor tampering and theft, a tampering alert component in the back-end is remotely wired to an accelerometer component that provides readings from the built-in accelerometer via an interface of type 'ACCEL'.

Policy-augmented composition: In contrast to the generic functionality offered by components, policies are used to tailor this functionality. For example, allowing scientists to set a specific level at which sensor readings should generate an alert. As a result of this, the basic application composition can be augmented with various types of policies as illustrated in Figure 3(b). An example policy for flood detection is provided in Listing 1, filtering readings at the source node and only allowing remote publication where the value exceeds a pre-defined threshold. As a result, the policy allows very specific customization of the flood alerting composition by the scientist, possibly depending on the time of season or the location of the sensor. It will not only prevent the publication of spurious alerts, but also conserve battery power. Similarly, an administrator can deploy a policy specifying an energy-aware alert reporting strategy for all environmental alerts inside the network. For instance, when the nodes are low on battery power everything must be stored locally, which is achieved through the publishing of a custom LOG event (Listing 2).

```
policy "Flood detection" {
   on PRESSURE as p; //PRESSURE contains parameter value
   if( p.value > 500 )
   then(
      allow p; //by default other PRESSURE events are blocked
   )
}
```

Listing 1. Example of flood detection policy

```
policy "Energy-aware reporting policy" {
   on ENVIRONMENTAL_ALERT as e;
   if( POWER_STATE == LOW )
   then(
      deny e; //do not propagate event
      publish(LOG, e.data[]); //but store it locally
   )
}
```

Listing 2. Example of energy management policy set by the administrator

Similarly, other possible unanticipated concerns might be addressed by the injection of policies at run-time. As a result, Figure 3(b) illustrates policies for conductivity detection, tampering detection, and methane detection which are possibly written by different users.

5 Evaluation and discussion

We have implemented and evaluated the performance of our policy-based programming paradigm on Java ME CLDC 1.1 compliant SunSPOT nodes [23] (180 MHz ARM9 CPU, 512 kB RAM, 4 MB flash, SQUAWK VM version 'RED-100104'). We investigate the overhead in terms of memory footprint, development overhead, and performance of policy evaluation and secure policy distribution.

5.1 Memory footprint and cost of change

As presented in Table 1, the footprint of the policy framework is small. The run-time consumes 28 kB of ROM, which represents 0,6 % of the total flash memory available on the SunSPOT. Dynamic memory requirements (RAM) for the policy framework are small, requiring only 0,2 % of the total available RAM. The footprint for the component model run-time and the example components is higher, but still small w.r.t. resource capabilities. The disparity between ROM and RAM requirements of a component can be explained by SunSPOT-specific overhead due to the inter-isolate RPC server and the establishment of proxies. Representing a policy is efficient, as the compact representation of the policies used in the case study only occupies 72 bytes of static memory on average. When combined with the distribution protocol header size of 15 bytes, it represents the

	Static footprint (ROM)	Dynamic footprint (RAM)	SLoC
Components:			
LooCI run-time	52 kB	37 kB	N/A
Conductivity Sensor	1.8 kB	26 kB	59
Methane Sensor	1.7 kB	26 kB	59
Pressure Sensor	1.7 kB	26 kB	59
Accelerometer	1.9 kB	26 kB	53
Environmental Alert	2.1 kB	26 kB	64
Logging Component	1.9 kB	27 kB	68
Policies:			
Framework run-time	28 kB	1 kB	N/A
Flood detection	64 bytes	284 bytes	7
Energy-aware reporting	102 bytes	440 bytes	8
Theft detection	65 bytes	292 bytes	7
Conductivity detection	63 bytes	296 bytes	7
Methane detection	62 bytes	286 bytes	7

Table 1. Comparison of memory requirements and development overhead

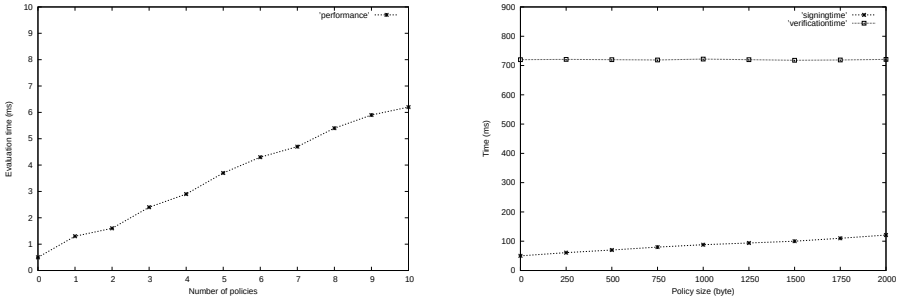
amount of data transmitted per policy from the gateway to an individual node. As a result, policy updates are lightweight compared to traditional component-based reconfiguration. Upon reception, the policy data is transformed into a Java object more suitable for efficient evaluation, requiring 320 bytes of RAM on average for the policies applied in the case study.

5.2 Development overhead

Table 1 also provides a quantitative assessment of development overhead in terms of Source Lines of Code (SLoC) of all components and policies. As can be seen, both paradigms are relatively compact and impose limited overhead on developers. The identical size of some artefacts listed in the table may be attributed to their simplicity and similarity (in all cases, components read a simple value from the SunSPOT Analog-to-Digital Converter (ADC) and transmit it over the event bus, whereas the policies do some simple filtering of spurious events).

5.3 Policy engine performance

Figure 4(a) illustrates the overhead of evaluating a number of policies against an event flowing between two components. The performance of policy evaluation includes the time to intercept and redirect the event to the policy engine (0.5 ms), followed by evaluation of matching policies, which was on average 0.5 ms per policy used in our case study. As can be seen from the figure, this scales linearly with the number of policies. Finally, it takes 6 ms on average to construct a policy from its compact representation into a suitable data structure, whereas 7420 ms are needed on average to initialize and start a regular component after deployment (due to registration with the LooCI run-time and establishment of inter-isolate proxies on the SunSPOT).



(a) Policy evaluation (b) Secure policy deployment

Fig. 4. Performance evaluation

5.4 Secure policy distribution performance

Subsequently, we investigate the overhead of secure policy distribution in terms of a varying length of `policy_data` as input. Since policy signing happens at the gateway, which in the case study is a standard Internet-connected PC located near the river, it can be done efficiently. We mainly concentrate on the time it takes to verify $MAC(K_{GW}, D)$ at the receiving node. For signing and verification, we use the SHA-1 hashing algorithm, which makes use of the pre-deployed public/private key scheme [22] of the SunSPOT. As can be seen from Figure 4(b), verifying the authenticity of the message can be done in a constant time order (i.e on average 720 ms). Only the time to calculate the signature slightly increases with the input size (which could hypothetically be large), albeit this signing happens at the resource-rich gateway tier. As policy updates in the field test involved less then 80 bytes of `policy_data` on average and 15 bytes of header data, we believe this overhead is still acceptable.

5.5 Lessons learned from the field test

Through the case, several points were uncovered. Firstly, the ability to deploy policies with different coverage was found advantageous. The distribution model allows a single policy, implementing a specific concern, to be deployed to different entities inside the network, ranging from an individual component interface on a single device to the entire network. Secondly, because of diversity in functional objectives, nature of the concerns, resource-constraints, and change cycle, WSN applications benefit from "mixing- and-matching" multiple programming paradigms. In particular, artefacts of varying granularity are incorporated to reduce the impact of change on the system. Both relatively coarse-grained components and lightweight, fine-grained policies are key elements of effective solutions. Base artefacts can reflect dynamism of the application functionality and the control rules govern the behaviour of that functionality. This paradigm mix also supports the conceptual decoupling of stakeholder abstractions. Ordinary

application users who often have little experience in programming, such as the ecologists in our case, had the ability to express how components developed by embedded systems experts should behave.

However, applying multiple programming paradigms next to each other must be done with care. One paradigm can introduce additional behaviour opaque to the other paradigm or create tight coupling between the various artefact developers, both leading to unwanted side effects. An operational model where for example the network administrator analyzes the interplay between both paradigms can mitigate this problem.

Regardless of the potential issues raised, it is our belief that the programming of WSN applications based on multiple paradigms holds great potential for large scale multi-actor scenarios such as the river monitoring application described in this paper. The approach respects both the skill-set of each actor and the resource constraints of the WSN.

6 Related work

In recent years, a number of programming abstractions [18] have been proposed to simplify WSN application development, including programming models adopting the principles of component-based engineering. Their associated update models can be classified as either static or dynamic. NesC [7], perhaps the best known component model in the WSN field, adopts an event-driven programming approach as the interaction model in combination with a static update model. Reconfiguration translates to deploying a monolithic image, allowing for whole-program analysis and optimization. Run-time reconfigurable component models such as RUNES [4], or OpenCOM [6] follow a dynamic update model, allowing the composition to be changed at run-time through the deployment of new components and modification of component bindings. Despite their benefits regarding reusability, components are still artefacts solely supporting coarse-grained modifications.

In order to use the component paradigm, additional support from the sensor operating system is equally needed. Modular updates at run-time can be provided, for example, by modular Virtual Machine (VM) solutions [13, 25] or component-oriented operating systems [20]. A major drawback of VMs applied in energy-constrained systems is that they interpret byte code as opposed to components executing native instructions, which results in overhead. Similar to this research on combining a lightweight policy-based paradigm with a more generic main development paradigm, Koshi et al. [13] describe a hybrid approach that efficiently combines VM byte code interpretation with native code execution.

Platon and Sei [19] emphasize the need for policy-based management of WSNs to provide for scalability of security. Marsh et al. [17] provide for a flexible and memory efficient policy specification language validating policy-based approaches for WSN management. Finger [26] provides support for the policy-based management of TinyOS [15] nodes in a relatively small footprint, though it offers no support of run-time reconfiguration of applications.

7 Conclusions and future work

This paper has made the case for a fine-grained, policy-driven approach to manage the diverse concerns of multiple actors involved in realistic WSN applications. The feasibility of this approach has been demonstrated through a prototype implementation on the SunSPOT platform, using an event-based component model as the base programming paradigm. We validated in a real world river monitoring scenario. As a result, the evaluation has shown that our approach is sufficiently lightweight and has clear benefits to be applied effectively in WSN environments.

In the short term, our future work will focus upon further investigation on the interplay of multiple co-existing programming paradigms. Furthermore, investigation of when to apply which paradigm is required as this might not always be always obvious for many goals. In this light, further collaboration with different WSN actors should give us the necessary insights.

Acknowledgement: This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the Research Fund K.U.Leuven, and is conducted in the context of the IWT-SBO-STADiUM project No. 80037 [11] and IWT-SBO-SymbioNets project No. 090062 [12].

References

1. D. Agrawal, S. Calo, K.-w. Lee, J. Lobo, and D. Verma. *Policy Technologies for Self-Managing Systems*. IBM Press, 2008.
2. L. S. Bai, R. P. Dick, and P. A. Dinda. Archetype-based design: Sensor network programming for application experts, not just programming experts. In *IPSN*, pages 85–96, 2009.
3. D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 175–188, New York, NY, USA, 2007. ACM.
4. P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis. Reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, 14(2):149–162, 2007.
5. G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems (4th ed.): concepts and design*. Addison-Wesley Publishing Co., Inc., Boston, MA, USA, 2001.
6. G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1–42, 2008.
7. D. Gay, P. Levis, R. V. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the 2003 PLDI*, pages 1–11, New York, USA, May 2003. ACM Press.
8. G. T. Heineman and W. T. Councill, editors. *Component-based software engineering: putting the pieces together*. Addison-Wesley Co., Boston, MA, USA, 2001.
9. D. Hughes, K. Thoelen, W. Horré, N. Matthys, P. J. del Cid Garcia, S. Michiels, C. Huygens, and W. Joosen. LooCI: A loosely-coupled component infrastructure

for networked embedded systems. In *Proceedings of the 7th International Conference on Advances in Mobile Computing & Multimedia*, pages 195–203. ACM, December 2009.
10. C. Huygens, D. Hughes, B. Lagaisse, and W. Joosen. Streamlining development for networked embedded systems using multiple paradigms. *IEEE Software*, September 2010.
11. IWT-SBO-STADiUM project No. 80037. Software technology for adaptable distributed middleware. http://distrinet.cs.kuleuven.be/projects/stadium/.
12. IWT-SBO-SymbioNets project No. 090062. Symbiotic networks. http://symbionets.intec.ugent.be/.
13. J. Koshy, I. Wirjawan, R. Pandey, and Y. Ramin. Balancing computation and communication costs: The case for hybrid execution in sensor networks. 6(8):1185 – 1200, 2008.
14. M. Kuorilehto, M. Hännikäinen, and T. D. Hämäläinen. A survey of application distribution in wireless sensor networks. *EURASIP J. Wirel. Commun. Netw.*, 2005(5):774–788, 2005.
15. P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. A. Brewer, and D. E. Culler. The emergence of networking abstractions and techniques in tinyos. In *Proc. 1st Symposium on NSDI*, pages 1–14, 2004.
16. A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, USA, 2002.
17. D. Marsh, R. Baldwin, B. Mullins, R. Mills, and M. Grimaila. A security policy language for wireless sensor networks. *Journal of Systems and Software*, 82(1):101–111, 2009.
18. L. Mottola and G. Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Computing Surveys*, 2010.
19. E. Platon and Y. Sei. Security software engineering in wireless sensor networks. *Progress in Informatics*, 5(1):1–19, 2008.
20. B. Porter and G. Coulson. Lorien: a pure dynamic component-based operating system for wireless sensor networks. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, pages 7–12. ACM, 2009.
21. J. Steffan, L. Fiege, M. Cilia, and A. Buchmann. Towards multi-purpose wireless sensor networks. In *Proceedings of the International Conference on Systems Communications*, pages 336–341, Washington, DC, USA, 2005.
22. Sun Microsystems. Sun SPOT security library. https://spots-security.dev.java.net/.
23. Sun Microsystems. Sun SPOT world. http://www.sunspotworld.com/.
24. M. Wang, J. Cao, J. Li, and S. K. Dasi. Middleware for wireless sensor networks: A survey. 23(3):305–326, 2008.
25. Y. Yu, L. Rittle, V. Bhandari, and J. LeBrun. Supporting concurrent applications in wireless sensor networks. In *Proc. of the 4th international conference on Embedded networked sensor systems*, pages 139–152, New York, NY, USA, 2006. ACM.
26. Y. Zhu, S. Keoh, M. Sloman, E. Lupu, N. Dulay, and N. Pryce. Finger: An Efficient Policy System for Body Sensor Networks. In *5th IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, September 2008.